REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1992	Technical Memorandum

4. TITLE AND SUBTITLE
The Syntax of DRAGOON: Evaluation and Recommendations

5. FUNDING NUMBERS
WU 505-64-10-02

6. AUTHOR(S)
C. Michael Holloway

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17028

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM-4385

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Several different ways to add linguistic support for object-oriented programming to the Ada programming language have been proposed and developed in recent years. DRAGOON is one such Ada extension. This paper describes the DRAGOON syntax for classes, objects, and inheritance, and it evaluates the syntax against the following five criteria: readability, writeability, lack of ambiguity, ease of translation, and consistency with existing Ada syntax. The evaluation reveals several deficiencies in the notation. The paper concludes with a proposal for a revised syntax that corrects these deficiencies.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada; Programming languages; DRAGOON; Syntax	16
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298(Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

NASA-Langley, 1992

Abstract

Several different ways to add linguistic support for object-oriented programming to the Ada programming language have been proposed and developed in recent years. DRAGOON is one such Ada extension. This paper describes the DRAGOON syntax for classes, objects, and inheritance, and it evaluates the syntax against the following five criteria: readability, writeability, lack of ambiguity, ease of translation, and consistency with existing Ada syntax. The evaluation reveals several deficiencies in the notation. The paper concludes with a proposal for a revised syntax that corrects these deficiencies.

1. Introduction

The Ada programming language was designed in the late 1970's and early 1980's (Nyberg 1989) before object-oriented programming became popular; as a result, Ada provides little support for these techniques (Anon. 1989a). In recent years, several groups have attempted to improve Ada support for object-oriented programming by developing extensions to the language.

Software Productivity Solutions, Inc., is currently marketing an Ada extension called Classic-Ada (Donaldson 1989; Anon. 1989b), which is based on the language Smalltalk (Goldberg and Robson 1983). Harris Corporation has developed a language for their internal use called InnovAda (Simonian and Crone 1988) that is based on Flavors (Moon 1986) (an object-oriented extension to Lisp that has been made obsolete by the Common Lisp Object System (Bobrow et al. 1988)). Several other approaches have been described also (Forestier, Fornarino, and Franchi-Zannettacci 1989; Winkler 1990).

The most widely praised Ada extension (Johnson 1990) is the programming language Distributable Reusable Ada Generated from an Object-Oriented Notation (DRAGOON) (Di Maio et al. 1989; Genolini, Di Maio, and De Michele 1990; Atkinson et al. 1991), which is being developed by TXT Ingegneria Informatica in Italy as part of the Esprit project (a joint effort of several European governments, companies, and universities to significantly improve software quality). DRAGOON is based on Eiffel (Meyer 1988) and provides full syntactic and semantic support for object-oriented programming.

This paper describes and evaluates the syntax of the DRAGOON extensions to Ada and proposes improvements to it. The rest of the paper is organized as follows: Section 2 gives necessary background information; section 3 describes the current syntax for DRAGOON and gives examples of its use; section 4 explains the criteria used to evaluate the syntax; section 5 evaluates the syntax and lists the deficiencies found in it; section 6 provides recommendations on how to improve the syntax; and section 7 gives concluding remarks.

2. Background Information

This section provides background information needed for understanding the remainder of the paper. Object-oriented programming terms and concepts are defined, and the notation used for describing the DRAGOON syntax is explained. Since the reader is assumed to be familar with Ada, an overview of Ada is not given.

2.1. Object-Oriented Programming Terms and Concepts

Reading the literature reveals that no universally adopted terminology exists for describing object-oriented programming concepts (Wirfs-Brock and Johnson 1990). The three most widely

known object-oriented programming languages—Smalltalk, C++ (Stroustrup 1986; Ellis and Stroustrup 1990), and Eiffel—often use different terms to refer to the same concepts or the same term to refer to different concepts. The terms and definitions below are based on Booch (1991) and Wegner (1990) and will be used in the rest of this paper.

An *object* is a software entity that consists of a set of state data and a set of operations on that data. Another common name for an object is an *instance*. The state data of an object are composed of its *instance variables*; the operations permitted on an object are its *methods*.

A *class* is a template for building similar objects. A class specifies the instance variables and methods that all objects belonging to the class will have. Together, these instance variables and methods are called the *features* of the class.

A class or subprogram that contains or uses an instance of another class is called a *client* of that class. Methods that may be used by a client are said to be *visible*. Methods that may be used only within the class in which they are defined are called *internal* methods. The visible methods compose the class *interface* or *specification*. Instance variables, internal methods, and the code for visible methods compose the class *implementation* or *body*.

Classes can share interfaces and implementations through *inheritance*, thus forming a class *hierarchy*. Within a hierarchy, a *descendant* class can inherit instance variables and methods from one or more *ancestor* classes. A hierarchy in which a descendant class can have more than one ancestor class uses *multiple inheritance*; a hierarchy in which a descendant class can have only one ancestor class uses *single inheritance*.

A class may specify the interface for some methods without providing an implementation for them. Such methods are called *deferred methods*, and the class containing them is called an *abstract* class. An abstract class cannot have any instances. Descendant classes can implement the given interface for the deferred methods of an abstract ancestor. A class that provides an implementation for every method in its interface is called a *concrete* class.

2.2. Syntax Notation

The published papers on DRAGOON do not give a formal definition of the syntax of the language; instead, they illustrate it with specific examples. The definitions used in the paper were developed by the author after a careful study of the published examples. The notation is the same as that used in describing the context-free syntax of Ada (Nyberg 1989). This notation is a variation of the well-known Backus-Naur Form (Pratt 1984), with the following rules:

- Lowercase words, which may include embedded underscores, denote syntactic categories; for example, `class_declaration`.

- Boldface words and symbols denote literal words and symbols. For example,

  class ;

  denotes the reserved word **class** followed by a literal semicolon.

- The symbol **::=** denotes a definition of a syntactic category. For example,

  ```
  class_declaration ::= class_specification ;
  ```

  defines the syntactic category `class_declaration` to consist of a `class_specification` followed by a semicolon.

- A vertical bar (|) separates alternative items. For example,

  ```
  method_modifier ::= completed | redefined
  ```

  means that a `method_modifier` may be either the reserved word **completed** or the reserved word **redefined**.

- Square brackets ([and]) surround an optional item. For example,

  ```
  method_specification ::= subprogram_specification [defer_part]
  ```

  means that a `method_specification` may, or may not, contain a `defer_part`.

- Braces ({ and }) surround an item that may appear zero or more times. For example,

  ```
  identifier_list ::= identifier {, identifier}
  ```

  defines an `identifier_list` to be an `identifier` followed by zero or more comma-separated `identifier`s.

- A syntactic category that begins with an italicized part is equivalent to the syntactic category without the italicized part. The italicized part is used to convey semantic information. For example, syntactically *class*`_identifier_list` is equivalent to `identifier_list`; the *class* prefix means that the language semantics requires that the identifiers denote names of classes.

- Only new and modified syntactic categories are defined; any category for which a definition is not specifically given has the same definition as it does in Ada.

The notation used in specific examples is the same as that used in the examples in the Ada definition. Specifically, reserved words are written in lowercase letters, and identifiers are written in uppercase letters with embedded underscores. However, this is only a convention; DRAGOON, like Ada, is insensitive to case. The examples given are meant to illustrate the syntax of DRAGOON only; they do not necessarily represent good programming style.

3. DRAGOON Syntax

This section describes the DRAGOON syntax for classes, objects, and inheritance. Minor aspects of the syntax, which would have no impact on the evaluation, are ignored.

The discussion, which concentrates on syntactic issues alone, neither provides an overview of the entire DRAGOON language nor discusses semantic issues. Also, DRAGOON allows classes to be parameterized, concurrent, and distributed, but these aspects of the language are not discussed. The syntax for parameterized classes is not described in any available published literature. Concurrent and distributed classes are not discussed because doing so would greatly complicate the discussion without significantly affecting the conclusions.

3.1. Classes

The syntactic structure of a class definition in DRAGOON is analogous to the syntactic structure of a package definition in Ada. Specifically, DRAGOON requires physical separation of a class interface and its implementation. The language also adopts the Ada nomenclature; the interface of a class is called its *specification*, and the implementation of a class is called its *body*.

3.1.1. Class specification. A DRAGOON class specification resembles an Ada package specification with the following three differences:

1. The DRAGOON reserved word **class** replaces the reserved word **package**.
2. The DRAGOON reserved word **introduces** is added.
3. Only method definitions and inheritance (see section 3.3) are allowed in a class specification.

Like a package specification, a class specification is a library unit. It may include **with** clauses to obtain visibility of classes or packages defined separately, and it may be compiled separately from the corresponding class body. Within a class specification, the syntactic form of a method declaration is identical to the syntactic form of an Ada subprogram declaration within a package specification.

The syntax for a class specification is given below:

```
library_unit ::=
     subprogram_declaration     |     package_declaration
   | generic_declaration         |     generic_instantiation
   | subprogram_body             |     class_declaration

class_declaration ::= class_specification ;

class_specification ::=
    class identifier is
        [ introduces_specification ]
    end [class_simple_name]

introduces_specification ::=
    introduces
        method_declaration
        { method_declaration}

method_declaration ::= method_specification ;

method_specification ::= subprogram_specification
```

Example 1 shows a simple class specification (Atkinson et al. 1991). This class defines an interface for a simple one-item buffer class called `UNI_BUFFER`; the interface consists of the two methods `PUT` and `GET`. The buffer accepts and makes available entities of type `ITEM` defined in the package `SIMPLE`.

```
with SIMPLE;
class UNI_BUFFER is
   introduces
     procedure PUT (I : in SIMPLE.ITEM);   -- method definition
     procedure GET (I : out SIMPLE.ITEM);  -- method definition
end UNI_BUFFER;
```

Example 1. A class specification.

3.1.2. Class body. A DRAGOON class body corresponds to an Ada package body even more closely than a class specification corresponds to a package specification. The only syntactic difference is the use of the reserved word **class** in place of **package**. No special restrictions limit what may appear within a class body. Each method defined in the class specification must have an implementation in the body, and any instance variables for the class must be defined in the body.

The following defines the syntax for a class body:

```
library_unit_body ::= subprogram_body | package_body | class_body

class_body ::=
  class body class_simple_name is
      [ declarative_part ]
  end [class_simple_name] ;
```

Example 2 shows a possible body for the `UNI_BUFFER` class (Atkinson et al. 1991). Except for the beginning reserved word, it appears identical to an Ada package body of the same name. Also, the instance variable declaration and the method bodies are identical syntactically to Ada variable declarations and subprogram bodies, respectively.

```
class body UNI_BUFFER is

    BUFFER : SIMPLE.ITEM;   -- instance variable

    procedure PUT (I  :  in SIMPLE.ITEM) is
    begin
      BUFFER := I;          -- method body
    end PUT;

    procedure GET (I : out SIMPLE.ITEM) is
    begin
      I := BUFFER;          -- method body
    end GET;

end UNI_BUFFER;
```

Example 2. A class body.

3.2. Objects

A DRAGOON object syntactically resembles an Ada variable. Specifically, the DRAGOON syntax for declaring an object of some class is identical to the Ada syntax for declaring a variable of some type. However, unlike an ordinary Ada variable, a DRAGOON object does not actually exist (that is, storage space is not allocated for it and its methods cannot be called) until explicitly created by either calling the special `CREATE` method (Atkinson et al. 1991) or by assigning to it an object for which the `CREATE` method has been called. The `CREATE` method is similar in effect to the **new** allocator for access types in Ada. The requirement for its use is a semantic issue, and thus it is not expressed in the syntax.

The DRAGOON notation for invoking a method resembles the Ada syntax for calling a subprogram, and it is consistent with the notation of most object-oriented programming languages. Outside of a class body, a method is invoked by giving its name and parameters prefixed by the name of an object belonging to a class in which the method is defined. Within a class body, an object name prefix is not usually required, and thus a method may be invoked by giving its name and parameters only.

The syntax for object declaration and method invocation is given below:

```
basic_declaration ::= as defined in Ada   |   class_object_declaration

class_object_declaration ::=
    identifier_list :  class_name ;

method_invocation ::=
    object_name.method_name [ actual_parameter_part ] ;

internal_method_invocation ::=
    method_name [ actual_parameter_part ] ;
```

Example 3 illustrates defining and creating an object, and also calling its methods. It also illustrates bypassing, through assignment, the need for an explicit `CREATE`.

```
with UNI_BUFFER, SIMPLE;
procedure USE_BUFFER is
    X : SIMPLE.ITEM;
    BUFF1 :  UNI_BUFFER;    -- define a UNI_BUFFER object
    BUFF2 :  UNI_BUFFER;    -- define another UNI_BUFFER object
begin
    BUFF1.CREATE;           -- create the object
    X := -- some appropriate value of type SIMPLE.ITEM
    BUFF1.PUT (X);          -- invoke the PUT method
    BUFF2 := BUFF1;         -- BUFF2 and BUFF1 now are the same object
    BUFF2.GET (X);          -- invoke the GET method
end USE_BUFFER:
```

Example 3. Object definition, creation, and use.

3.3. Inheritance

DRAGOON supports multiple inheritance for classes. The language allows a descendant class to modify methods from an ancestor and to add its own methods and variables. DRAGOON also allows an ancestor class to defer to its descendants the implementation of a method.

Syntactically, a descendant class lists its ancestors in its specification following the reserved word **inherits**. Inherited methods that are to be modified are listed after the reserved word **redefines**, and inherited methods that are to be completed (that is, given an implementation) are listed following the reserved word **completes**. A method may be completed only if it was specified as deferred (signified by the reserved words **is deferred**) by an ancestor.

The following defines the syntax for inheritance:

```
class_specification ::=
    class identifier is
        [ inherits_list ]
        [ redefines_list ]
        [ completes_list ]
        [ introduces_specification ]
    end [class_simple_name]

inherits_list ::=
    inherits
        class_identifier_list ;

redefines_list ::=
    redefines
        method_identifier_list ;

completes_list ::=
    completes
        method_identifier_list ;

method_specification ::= subprogram_specification [defer_part]

defer_part ::= is deferred
```

Example 4 illustrates the syntactic aspects of inheritance; it includes redefined, deferred, and completed methods (Di Maio et al. 1989). To avoid unnecessary complexity in the example, the class bodies are not shown.

The body of class PRINTER_DEVICE may define some instance variables and internal subprograms, but it cannot give bodies for either the RESET or PRINT methods, because these methods are deferred. The body of class DAISY_PRINTER must provide implementations for both methods because they are listed in its **completes** section.

Objects of class LASER_PRINTER have access to the methods RESET, PRINT, and GRAPHIC_PRINT via inheritance. The body of class LASER_PRINTER must contain implementations for the new method LOAD_FONT and for the redefined methods RESET (inherited from DAISY_PRINTER) and GRAPHIC_PRINT (inherited from GRAPHIC_DEVICE). It cannot have an implementation for the method PRINT because that method is inherited without redefinition from DAISY_PRINTER.

4. Criteria for Evaluation

The syntax of a programming language serves two primary purposes: it provides the notation for communication between programmers, and it provides the notation by which a programmer communicates information to a language processor. Because people are not machines, these two purposes often conflict. Since a language designer cannot create a syntax that fulfills both purposes simultaneously and fully, he must compromise. Pratt has listed four general criteria that a designer may use to guide and evaluate those compromises (Pratt 1984):

```
class PRINTER_DEVICE is
    introduces
        procedure RESET is deferred;
        procedure PRINT (F : in FILE) is deferred;
end PRINTER_DEVICE;

class DAISY_PRINTER is
    inherits
        PRINTER_DEVICE;
    completes
        RESET, PRINT;
end DAISY_PRINTER;

class GRAPHIC _DEVICE is
    introduces
        procedure GRAPHIC_PRINT (F : in FILE);
end GRAPHIC_DEVICE;

class LASER_PRINTER is
    inherits
        DAISY_PRINTER, GRAPHIC_DEVICE;
    redefines
        RESET, GRAPHIC_PRINT;
    introduces
        procedure LOAD_FONT;
end LASER_PRINTER;
```

Example 4. Multiple inheritance with redefined and deferred methods.

1. **Readability**—a programmer can deduce the underlying structure of the algorithms and data structures of a program by reading its source.
2. **Writeability**—a programmer can express algorithms and data structures naturally and concisely.
3. **Lack of ambiguity**—each syntactic construct has one and only one meaning.
4. **Ease of translation**—a program can be translated into an executable form cheaply and quickly.

The syntax of a particular programming language depends on the relative importance that its designers assign to satisfying each of these criteria. For example, the designers of the Ada programming language considered readability to be paramount (Nyberg 1989). In contrast, the designers of the `C` programming language considered writeability and ease of translation to be most important (Kernighan and Ritchie 1978). As a result, the syntax of Ada differs greatly from the syntax of `C`.

In developing or evaluating the notation of *an extension* to an existing language, one more criterion is important:

5. **Consistency with existing syntax**—the syntax obeys the conventions established in the base language.

If the notation of an extension is significantly different from the base notation—or worse, if its conventions and style conflict with those of the base notation—programmers will find the extension notation difficult to learn and use. This is true even if the extended syntax is, by itself, readable, writeable, free of ambiguity, and easy to translate.

5. Evaluation of DRAGOON Syntax

Evaluated against the above-mentioned criteria, the syntax of DRAGOON has two shortcomings:

1. Lists of method names, such as those required in a **redefines** or **completes** section, hinder readability, writeability, and ease of translation.
2. The overall style of the syntax seems to conflict with the style of the underlying Ada syntax.

The rest of this section explains in detail these two deficiencies of the syntax and illustrates both with examples.

5.1. Lists of Method Names

Section 3.3 discussed the syntax for redefined or completed methods; a list of method names follows the appropriate reserved word. The primary difficulty with this syntax is that it obscures the association between a method and the class in which it was defined. As a result, programs are less easy to read, translate, and write than they need to be.

To illustrate the problem, consider the **redefines** section of class `LASER_PRINTER` of example 4. Method `RESET` is inherited from class `DAISY_PRINTER` and method `GRAPHIC_PRINT` is inherited from class `GRAPHIC_DEVICE`, but this is not evident from the text of the class specification alone. To obtain this information, a programmer (or a language translator) must examine the class specifications for each ancestor class and look for the definition of a method with the appropriate name.

Lists of method names also prohibit a programmer from using the same name for a method within two or more classes that might have a common descendant. This hinders writeability since a programmer might have to invent different names simply to prevent possible, future name clashes. Name lists do not provide a way to resolve name clashes within the descendant, which is where the problem actually is.

Example 5 illustrates the difficulty. Class `C` inherits two methods named `P`, one from class `A` and one from class `B`. Within class `C` and any of its clients, a reference to `P` is ambiguous.

5.2. Differences in Style

The second deficiency in the described DRAGOON syntax is that its style does not conform fully to the style of the language on which it was based. At least three inconsistencies between the syntactic style of DRAGOON and Ada can be identified.

One conflict in style between the two languages is that DRAGOON does not follow the Ada convention of using different syntactic constructs to distinguish different entities. Specifically, a method specification is syntactically identical to a subprogram specification, but a method is not semantically identical to a subprogram. The DRAGOON syntax is not ambiguous (that is, a programmer or a translator can always determine from the context whether a particular specification defines a method or a subprogram), but it can be confusing. In contrast, Ada uses

different reserved words to specify a function and a procedure, although doing so is not strictly necessary either.

```
class A is
    introduces
        procedure P;
        procedure M (X : A);
end A;

class B is
    introduces
        procedure P;
        procedure N (Y : B);
end B;

class C is
    inherits A, B;
    -- is the method P that is visible to the clients of C inherited from
    -- class A or from class B?
end C;
```

Example 5. Conflicts in method names.

A second conflict in style is the DRAGOON sectioning of declarations. Ada does not have separate sections for the various types of declarations; variable, exception, subprogram, package, and task declarations generally may be interspersed in whatever manner seems best to a programmer. The only separate section in an Ada package is the private part, and it is there to help language translators only. DRAGOON violates this convention by having separate sections for inheritance, method definitions, method completion, and method redefinition.

The third conflict is the DRAGOON imposition of an order on sections. Ada places few restrictions on the order of declarations; the only requirement is that a name be defined before it is used. The sections of a DRAGOON class specification must follow a partial order: the **inherits** section must precede either a **redefines** or a **completes** section, and the **introduces** section must follow all these.

Taken together, these last two aspects of DRAGOON make a class specification resemble syntactically a Pascal program more than an Ada package specification. As an illustration, example 6 shows the basic structure of a DRAGOON class specification, a Pascal program, and an Ada package specification. The Ada package specification imposes very little structure on its contents; however, the DRAGOON class specification, like a Pascal program, imposes a fairly rigid structure on its contents.

6. Possible Improvements

Neither of the deficiencies in the DRAGOON syntax noted in the previous section are severe, and both can be eliminated by making three modifications to the syntax. The following two modifications are simple:

1. Introduce the new reserved word **method** to be used instead of **procedure** in method definitions. This change eliminates the first style conflict identified in section 5.2.
2. Eliminate the **introduces** section. This change partially addresses the second and third style conflicts discussed in section 5.2.

```
        DRAGOON                  Pascal                  Ada

class C is                program P                package P is
  inherits                  label                    <most anything>
    <only class names>        <only labels here>     < can go here >
  redefines                 const
    <only method names>       <only constants>     private
  completes                 type                     <most anything>
    <only method names>       <only type defs>       < can go here >
  introduces                var
    <only method specs>       <only var decls>     end P;
end C;                    begin
                            .  .  .
                          end.
```

Example 6. Structure of DRAGOON, Pascal, and Ada.

The result of making these two modifications is described by the following syntax:

```
class_declaration ::= class_specification ;

class_specification ::=
    class identifier is
        { method_or_inherit_declaration }
    end [class_simple_name]

method_or_inherit_declaration ::= method_declaration
    |   inherit_declaration

method_declaration ::= method_specification [defer_part] ;

method_specification ::=
    method identifier [formal_part]

defer_part ::= is deferred

class_body ::=
    class body class_simple_name is
        [ class_body_ declarative_part ]
    end [class_simple_name] ;

class_body_declarative_part ::=
    { basic_declarative_item } { class_body_later_declarative_item }

class_body_later_declarative_item ::=
    later_declarative_item   |   method_body
```

```
method_body ::=
    method_specification is
        [ declarative_part ]
    begin
        sequence_of_statements
    end [method_simple_name] ;
```

Example 7 shows how the class specification of example 1 and the class body of example 2 would be rewritten using the modified syntax.

```
with SIMPLE;
class UNI_BUFFER is
    method PUT (I : in SIMPLE.ITEM);
    method GET (I : out SIMPLE.ITEM);
end UNI_BUFFER;

class body UNI_BUFFER is

    BUFFER : SIMPLE.ITEM; -- instance variable

    method PUT (I : in SIMPLE.ITEM) is
    begin
        BUFFER := I;
    end PUT;

    method GET (I : out SIMPLE.ITEM) is
    begin
        I := BUFFER;
    end GET;

end UNI_BUFFER;
```

Example 7. Revised class specification and body.

These two modifications neither address the problem caused by lists of method names (section 5.1) nor fully resolve the conflicts in syntactic style between DRAGOON and Ada. To solve the remaining shortcomings of the DRAGOON syntax, a third modification is needed. Specifically, the **inherits**, **redefines**, and **completes** sections must be eliminated and replaced by *inheritance clauses*, which may appear interspersed with method definitions in a class specification.

A separate inheritance clause is used for each ancestor class. The clause begins with the reserved word **inherit** followed by the name of the ancestor class. After the ancestor name, the methods from the ancestor class that are to be redefined or completed are fully specified. Thus, the relationship between a method and its defining class is explicit; neither a programmer nor a translator have to perform a search to discover the relationship. The clause is terminated by the reserved words **end inherit**. If ancestor methods are inherited without modification, the terminating phrase is not needed.

The following syntax describes the inheritance clause. In this syntax, an inheritance clause containing no completed method redefinitions is called a *simple* inheritance clause;

an inheritance clause containing method redefinitions or completions is called a *compound* inheritance clause. Thus,

```
inherit_declaration ::= inherit_clause ;
inherit_clause ::= simple_inherit_clause   |   compound_inherit_clause
simple_inherit_clause ::= inherit class_simple_name
compound_inherit_clause ::=
    inherit class_simple_name is
        { method_modification_declaration }
    end inherit [ class_simple_name ]
method_modification_declaration ::=
    method_specification is method_modifier ;
method_modifier ::= completed   |   redefined
```

Example 8 shows the classes of example 4 rewritten to conform to the proposed new syntax. With the new syntax, a programmer can see immediately, without visually searching back through the program text, in which class each completed and redefined method was originally defined.

```
class PRINTER_DEVICE is
    method RESET is deferred;
    method PRINT (F : in FILE) is deferred;
end PRINTER_DEVICE;

class DAISY_PRINTER is
    inherit PRINTER_DEVICE is
        method RESET is completed;
        method PRINT (F : in FILE) is completed;
    end inherit PRINTER_DEVICE;
end DAISY_PRINTER;

class GRAPHIC_DEVICE is
    method GRAPHIC_PRINT (F : in FILE);
end GRAPHIC_DEVICE;

class LASER_PRINTER is
    inherit DAISY_PRINTER is
        method RESET is redefined;
    end inherit DAISY_PRINTER;

    method LOAD_FONT; -- note that the order does not matter

    inherit GRAPHIC_DEVICE is
        method GRAPHIC_PRINT (F : in FILE) is redefined;
    end inherit GRAPHIC_DEVICE;

end LASER_PRINTER;
```

Example 8. Printer example rewritten.

Another advantage of the inheritance clause is that it provides a context for renaming methods whose original name conflicts with that of another method. A construct for such renaming is not included in the syntax given above, but one can be added easily. Example 9, which is a modification of example 5, illustrates one possible solution.

```
class A is
    method P;
    method M (X : A);
end A;

class B is
    method P;
    method N (Y : B);
end B;

class C is
    inherit A is
        method CP renames P;
    end inherit A;
    inherit B; -- no need to rename P, since conflict already resolved
end C;
```

Example 9. Method name conflicts resolved.

7. Concluding Remarks

This paper has described and evaluated the DRAGOON syntax for classes, objects, and inheritance. The evaluation revealed two deficiencies in the syntax. First, lists of method names, such as those required in a **redefines** or **completes** section, hinder the readability, writeability, and ease of translation. Second, the overall style of the syntax seems to conflict with the style of the underlying Ada syntax. Neither of these deficiencies is severe, and both can be eliminated without difficulty.

This paper proposes the following three modifications to the syntax, which the author believes can correct the current deficiencies:

1. In method definitions, introduce the new reserved word **method** to be used instead of **procedure**.
2. Eliminate the **introduces** section.
3. Replace the **inherits**, **redefines**, and **completes** sections by an *inheritance clause*, which makes explicit the relationship between inherited methods and their defining classes.

Implementing these changes should make DRAGOON programs easier to understand and write than they are currently, and this simplification should result in a greater consistency between the syntactic styles of DRAGOON and Ada.

NASA Langley Research Center
Hampton, VA 23665-5225
June 5, 1992

8. References

Anon. 1989a: *Ada 9X Project Report—Ada 9X Project Requirements Workshop.* Office of the Under Secretary of Defense for Acquisition.

Anon. 1989b: *Classic-Ada User's Manual.* Software Productivity Solutions.

Atkinson, Colin; Goldsack, Stephen; Di Maio, Andrea; and Bayan, Rami 1991: Object-Oriented Concurrency and Distribution in DRAGOON. *J. Object-Oriented Program.,* vol. 4, no. 1, pp. 11–20.

Bobrow, Daniel G.; DeMichiel, Linda G.; Gabriel, Richard P.; Keene, Sonya E.; Kiczales, Gregor; and Moon, David A. 1988: Common Lisp Object System Specification. *SIGPLAN Not.,* vol. 23, Special Issue, pp. 1-1–1-48.

Booch, Grady 1991: *Object Oriented Design With Applications.* Benjamin/Cummings Publ. Co., Inc.

Di Maio, Andrea; Cardigno, Cinzia; Bayan, Rami; Destombes, Catherine; and Atkinson, Colin 1989: DRAGOON: An Ada-Based Object Oriented Language for Concurrent, Real-Time, Distributed Systems. *Ada: The Design Choice—Proceedings of the Ada-Europe Conference*, Cambridge Univ. Press, pp. 39–48.

Donaldson, C. M. 1989: Dynamic Binding and Inheritance in an Object-Oriented Ada Design. *Ada: The Design Choice—Proceedings of the Ada-Europe Conference*, Cambridge Univ. Press, pp. 16–25.

Ellis, Margaret A.; and Stroustrup, Bjarne 1990: *The Annotated C++ Reference Manual.* Addison-Wesley Publ. Co.

Forestier, J. P.; Fornarino, C.; and Franchi-Zannettacci, P. 1989: Ada++—A Class and Inheritance Extension for Ada. *Ada: The Design Choice—Proceedings of the Ada-Europe Conference*, Cambridge Univ. Press, pp. 3–15.

Genolini, S.; Di Maio, A.; and De Michele, M. 1990: DRAGOON and Ada: The Wedding of the Nineties. *Proceedings of the Seventh Washington Ada Symposium*, Joseph P. Johnson, ed., Assoc. for Computing Machinery, Inc., pp. 245–254.

Goldberg, Adele; and Robson, David 1983: *Smalltalk-80, The Language and Its Implementation.* Addison-Wesley Publ. Co.

Johnson, Joseph P., ed. 1990: *Proceedings of the Seventh Washington Ada Symposium.* Assoc. for Computing Machinery, Inc.

Kernighan, Brian W.; and Ritchie, Dennis M. 1978: *The C Programming Language.* Prentice-Hall, Inc.

Meyer, Bertrand 1988: *Object-Oriented Software Construction.* Prentice Hall, Inc.

Moon, David A. 1986: Object-Oriented Programming With Flavors. *SIGPLAN Not.,* vol. 21, no. 11, pp. 1–8.

Nyberg, Karl A., ed. 1989: *The Annotated Ada Reference Manual.* ANSI/MIL-STD-1815A-1983 (Annotated).

Pratt, Terrence W. 1984: *Programming Languages—Design and Implementation*, Second ed. Prentice-Hall, Inc.

Simonian, Richard; and Crone, Michael 1988: InnovAda: True Object-Oriented Programming in Ada. *J. Object-Oriented Program.*, vol. 1, no. 4, pp. 14–21.

Stroustrup, Bjarne 1986: *The C++ Programming Language.* Addison-Wesley Publ. Co.

Wegner, Peter 1990: Concepts and Paradigms of Object-Oriented Programming—Expansion of Oct. 4 OOPSLA-89 Keynote Talk. *ACM OOPS Messenger*, vol. 1, no. 1, pp. 7–87.

Winkler, Jürgen F. H. 1990: Adding Inheritance to Ada. *Proceedings of the Seventh Washington Ada Symposium*, Joseph P. Johnson, ed., Assoc. for Computing Machinery, Inc., p. 241–244.

Wirfs-Brock, Rebecca J.; and Johnson, Ralph E. 1990: Surveying Current Research in Object-Oriented Design. *Commun. ACM*, vol. 33, no. 9, pp. 104–124.